نام و نام خانوادگی: فرزام کوهی رونقی **شماره دانشجویی:** 401106403

آزمون میانترم طراحی سیستم‌های دیجیتال

سوال 1 :

سؤال ۱:

یک ماژول وریلاگ با نام STACK_BASED_ALU برای اعداد صحیح علامت‌دار n بیتی طراحی کنید. ماژول باید دارای ورودی‌ها و خروجی‌های زیر باشد:

input_data: ورودی n بیتی

output_data: خروجی n بیتی

opcode: opcode ورودی 3-bit (مشخص‌کردن عملیاتی که باید انجام شود)

overflow: خروجی بیت سرریز

(توجه: اگر خروجی/ورودی دیگری هم نیاز است به ماژول بیفزایید)

ماژول باید از عملیات زیر پشتیبانی کند:

Opcode '100': Addition
Opcode '101': Multiply
Opcode '110': PUSH
Opcode '111': POP
Opcode '0xx': No Operation (the term 'x' means 0 or 1)

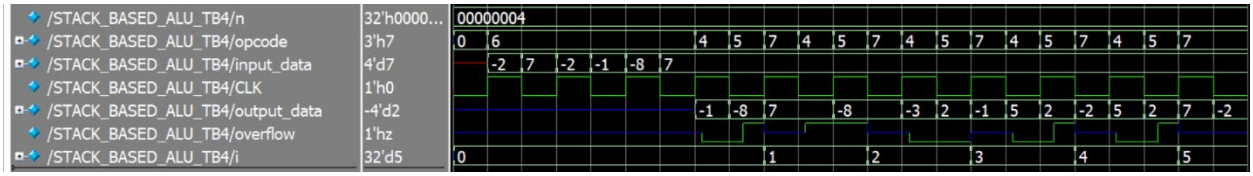
عملوندهای Opcode ضرب و جمع دو عدد بالایی پشته است. دقت کنید نتیجه عملیات ضرب/جمع در خروجی ماژول در دسترس خواهند بود و تغییری در پشته ایجاد نخواهند کرد.

ابتدا ماژول STACK_BASED_ALU در فایل Question1.v ساخته شده است. این ماژول شامل ورودی‌های CLK, input_data, opcode است و خروجی‌های output_data, overflow را شامل می‌شود. ورودی input_data و خروجی output_data، n بیتی هستند و با پارامتر n مشخص می‌شوند. این ماژول شامل یک آرایه از وکتور است که نقش استک را بازی می‌کند (stack نام دارد) و یک stack_pointer داریم که نشاندهنده بالاترین اندیس در استک ماست. دقت شود که ALU من بصورت CLOCK_BASED کار می‌کند و عملیات‌هایش را هم در لبه بالارونده و هم لبه پایین‌رونده انجام می‌دهد. آپکدها مشابه فایل داک میانترم تنظیم شده است. دقت شود که در دستورات Addition و Multiply در صورتی که در استک کمتر مساوی 1 عدد موجود باشد، خروجی z برمی‌گرداند. در حالتی که تعداد اعداد درون استک، بیشتر مساوی 2 باشد، حاصل عملیات را روی دو عدد بالای استک، در output_data می‌ریزد. دستور push عدد موجود در input_data را در استک پوش می‌کند (دقت شود که عمق استک یا همان stack_depth که نشانگر تعداد بیت‌های لازم برای آدرس عدد در استک است، برابر 10 قرار دادم و می‌توانستم آن را پارامتر بگذارم ولی بنظر 2^{10} خانه برای استک کافی است.)

برای دستور Push، تا جایی که استک پر نشده (شرط پر نشدن در کد بررسی شده) امکان پوش وجود دارد و برای دستور pop، اگر استک خالی باشد، output_data برابر z قرار می‌گیرد. بیت overflow برای جمع در حالتی 1 می‌شود که دو عدد operand، هر دو منفی و حاصل مثبت باشد یا اینکه هر دو مثبت باشند و حاصل منفی شود. برای محاسبه بیت overflow در ضرب، از یک متغیر با 2n بیت به نام result استفاده کردم که حاصل ضرب واقعی را در 2n بیت محاسبه کرده و اگر n+1 بیت سمت چپ آن، برابر کنار هم قرار دادن n بیت رقم سمت چپ result و رقم آخر output_data بود، یا اینکه علامت حاصل ضرب، خلاف انتظار بود، overflow = 1 در غیراینصورت overflow = 0. در دستورات push و pop نیز، overflow = z.

الف) برای طراحی خود testbench نوشته و آن را برای nهای ۴، ۸، ۱۶ و ۳۲ شبیه‌سازی کنید تا از صحت کارکرد آن مطمئن شوید. در آزمون خود بررسی صحت بیت‌سرریز را فراموش نکنید (۴۰ نمره).

تست صحت برای 4 بیت، در فایل Question1_testbench4 موجود است. شکل waveform آن به مانند زیر است:



عملیات‌های انجام شده همانطور که از opcode و سایر سیگنال‌ها مشخص است، به مانند زیر می‌باشد:

Push -2

Push 7

Push -2

Push -1

Push -8

Push 7

Add → 7 + (-8) = -1

Mul → 7 * (-8) = -56 → -8 // overflow

Pop → 7

Add → -8 + (-1) = -9 → 7 // overflow

Mul → -8 * (-1) = 8 → -8 // overflow

Pop → -8

Add → -1 + (-2) = -3

Mul → (-1) * (-2) = 2

Pop → -1

Add → 7 + (-2) = 5

Mul → 7 * (-2) = -14 → 16 // overflow

Pop → -2

Add → 7 + (-2) = 5

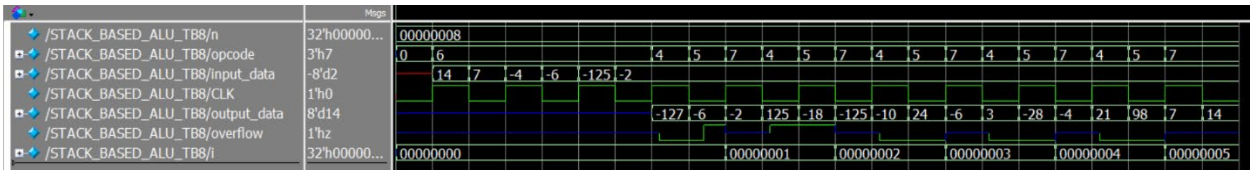
Mul → 7 * (-2) = -14 → 2 // overflow

Pop →7

Pop →-2

خروجی‌های ذکرشده در سیگنال output_data و overflow قابل مشاهده هستند.

تست صحت برای 8 بیت در فایل Question1_testbench.v و ماژول STACK_BASED_ALU_TB8 موجود است، شکل waveform آن به صورت زیر است:



دستورات اجراشده نیز به مانند زیر است:

Push 14

Push 7

Push -4

Push -6

Push -125

Push -2

Add → -2 + (-125) = -127

Mul → -2 * (-125) = 250 → -6 // overflow

Pop → -2

Add → -125 + (-6) = -131 → 125 // overflow

Mul → -125 * (-6) = 750 → -18 // overflow

Pop → -125

Add → -6 + (-4) = -10

Mul → -6 * (-4) = 24

Pop → -6

Add → -4 + 7 = 3

Mul → -4 * 7 = -28

Pop → -4

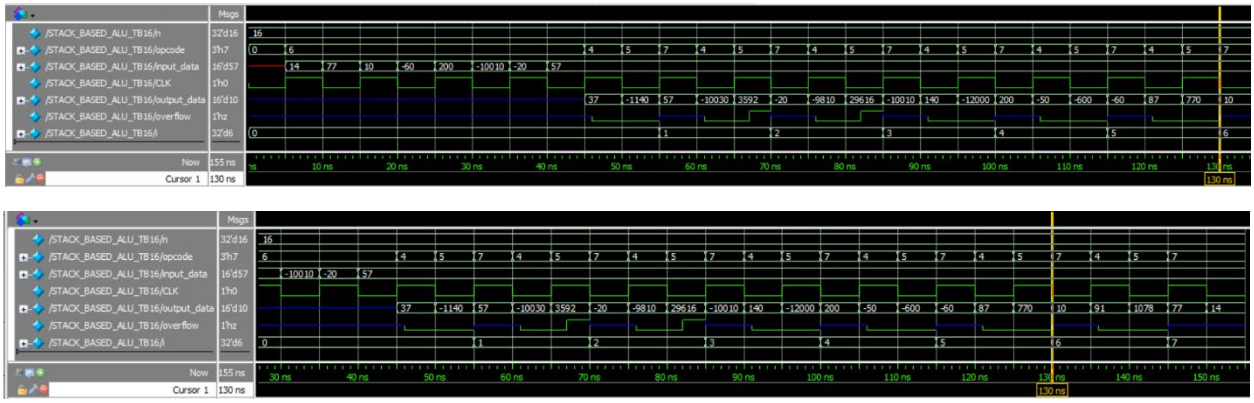
Add → 7 + 14 = 21

Mul → 7 * 14 = 98

Pop → 7

Pop → 14

تست صحت برای 16 بیت در فایل Question1_testbench16.v موجود است. شکل waveform مانند زیر است: (دو عکس ادامه هم هستند)



Push 14

Push 77

Push 10

Push -60

Push 200

Push -10010

Push -20

Push 57

Add → 57 + (-20) = 37

Mul → 57 * (-20) = -1140

Pop → 57

Add → -10010 + (-20) = -10030

Mul → -10010 * (-20) = 200200 → 3592 // overflow

Pop → -20

Add → (-10010) + 200 = -9810

Mul → (-10010) * 200 = -2002000 → 29616 // overflow

Pop → -10010

Add → 200 + (-60) = 140

Mul → 200 * (-60) = -12000

Pop → 200

Add → -60 + 10 = 50

Mul → -60 * 10 = -600

Pop → -60

Add → 10 + 77 = 87

Mul → 10 * 87 = 870

Pop → 10

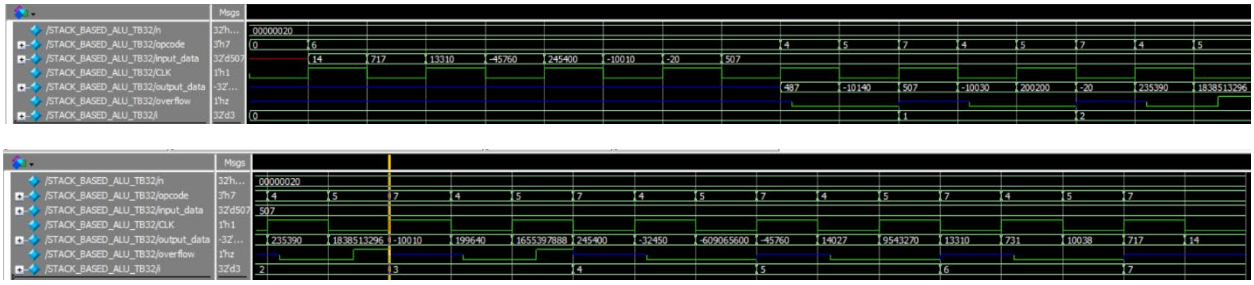
Add → 14 + 77 = 91

Mul → 14 * 77 = 1078

Pop → 77

Pop → 14

تست صحت برای 32 بیت در فایل Question1_testbench32.v موجود است. تصویر waveform آن به صورت زیر است: (تصویر دوم از خط زرد به بعد، ادامه تصویر اول است.)



سبک دستورات دقیقا مشابه 3 بخش قبل است صرفا اعداد تغییر کرده، به منظور جلوگیری از اتلاف وقت، صرفا اشاره می‌کنم که اعداد به ترتیب چه بوده اند:

14, 717, 13310, -45760, 245400, -10010, -20, 507

این اعداد به ترتیب در استک، پوش شده اند و بعد از آن در هر گام، ابتدا عملیات جمع و سپس ضرب و بعد pop رخ داده و اینکار ادامه می‌یابد تا زمانی که استک خالی شود. حاصل جمع و ضرب و pop در output_data و سیگنال overflow نیز نمایانگر اورفلو شدن جمع یا ضرب است.

ب) با استفاده از ماژول STACK_BASED_ALU ماژول دیگری نوشته که بتواند عباراتی مانند عبارات زیر را ابتدا به پسوندی تبدیل کند و سپس نتیجه آن را محاسبه کند (۲۰ نمره).

2 * 3 + (10 + 4 + 3) * -20 + (6 + 5)

برای این بخش، ابتدا نیاز به ماژولی حس می‌شد که بتواند یک عبارت ریاضی infix را به یک عبارت ریاضی postfix تبدیل کند، بنابراین ابتدا به دنبال الگوریتمی مناسب برای تبدیل یک عبارت ریاضی infix به postfix گشتم. لینک زیر را پیدا کردم:

https://www.andrew.cmu.edu/course/15-200/s06/applications/ln/junk.html#:~:text=To%20convert%20an%20infix%20expression,same%20expression%20in%20prefix%20notation.

در این لینک، الگوریتمی توضیح داده شده که با یک رشته ورودی به نام input که همان عبارت infix ماست و یک رشته خروجی به نام output که همان خروجی نهایی postfix شده ماست و یک استک و یک جدول به نام جدول precedence ها در استک و اینپوت، عملیات تبدیل عبارت infix به postfix را انجام می‌دهد. به منظور جلوگیری از خروج از هدف اصلی آزمون، از توضیح این الگوریتم صرف نظر می‌کنم اما الگوریتم به طور کامل در لینک داده شده توضیح داده شده است.

حال این الگوریتم را در ابتدا در فایلی به نام InfixToPostfis.v با نام ماژولی با همین نام، پیاده سازی کردم. این ماژول دو پارامتر به عنوان n, length دارد. n همان حداکثر تعداد بیت‌های اعداد ورودی است. Length نیز مجموع تعداد operand ها و operation ها و پرانتز باز و بسته است. برای مثال در عبارت زیر:

5 – 4 * (2 + 1 + -2)

Length برابر 11 است (توجه شود که 2-، یکبار در length شمرده می‌شود). یک ورودی infix دارم که همان عبارت infix است و خروجی‌ها عبارتند از postfix که همان عبارت postfix ماست، token که در هر گام حلقه، نشان‌دهنده operand یا operation یا پرانتز باز یا بسته ای است که در حال بررسی آن هستیم و top_of_stack برای زمانی است که به عنصر بالای استک نیاز داریم. تعدادی نیز سیگنال برای دیباگ مدار گذاشتم، مانند CLK_out, In_out, out_out, opcode_out که هر کدام طبق اسمشان معلوم است چه چیزی هستند، صرفا باید بگویم که in_out همان input_data برای استک و out_out همان output_data برای استک است.

از آنجا که کلاک سایکل تایم در این ماژول برابر 20 است، دو دسته زمان در تئوری تعریف کرده‌ام:

زمان‌های به صورت 10k+5 که در وسط میان دو لبه از یک کلاک هستند، که به آنها زمان !reset گفته ام (در کامنت‌های کدم قابل مشاهده است) و زمان‌های به صورت 10k’ که محل انجام کار است و در واقع یک لبه کلاک در آن رخ می‌دهد و چون ماژول STACK_BASED_ALU ما به لبه (چه بالارونده و چه پایین‌رونده) حساس است، در این لبه‌ها کار صورت می‌گیرد. بنابراین زمان‌بندی بخش‌های مختلف کدم با این فرضیات صورت گرفته و سعی شده بعد از پایان هر بخش (مانند یک if statement یا یک بلاک خاص) به یک زمان استراحت به صورت 10k+5 برسیم.

دقت شود که ورودی و خروجی ما به صورت یک وکتور 3 بعدی است:

input [1:0][length-1:0][n-1:0] infix

output reg [1:0][length-1:0][n-1:0] postfix

بعد اول آن، اگر 0 باشد نمایانگر مقدار operand یا operator یا پرانتز باز یا بسته است و اگر 1 باشد نمایانگر نقش آن است، به طوری که اگر اندیس 1، مقدار 0 داشته باشد، یعنی با operand سر و کار داریم و اگر 1 باشد با operator یا پرانتز. برای درک بهتر به مثال زیر توجه کنید!

فرض کنید که می‌خواهیم عبارت ریاضی (1 * 3) + 2 را در infix ورودی دهیم. ورودی بدین شکل خواهد بود:

Input[0][0] = 2

Input[1][0] = 0

این یعنی اولین بخش رشته infix ما، عدد 2 است، مقدار آن از input[0][0] = 2 می‌آید و اینکه عدد هست یا خیر، از input[1][0] می‌آید که چون 0 است، یعنی با عدد سر و کار داریم.

Input[0][1] = ‘+’ = 43

Input[1][1] = 1

این یعنی دومین بخش رشته infix ما، علامت + است که مقدار اسکی آن برابر 43 می‌باشد و اینکه این بخش، یک operand یا پرانتز است چونکه input[1][1] = 1.

Input[0][2] = ‘(‘ = 40

Input[1][2] = 1

این نیز یعنی بخش سوم رشته، یک علامت) است که مقدار اسکی آن برابر 40 بوده و چونکه Input[1][2] = 1، باید به آن به چشم یک operand یا پرانتز نگاه کنیم و نه عدد.

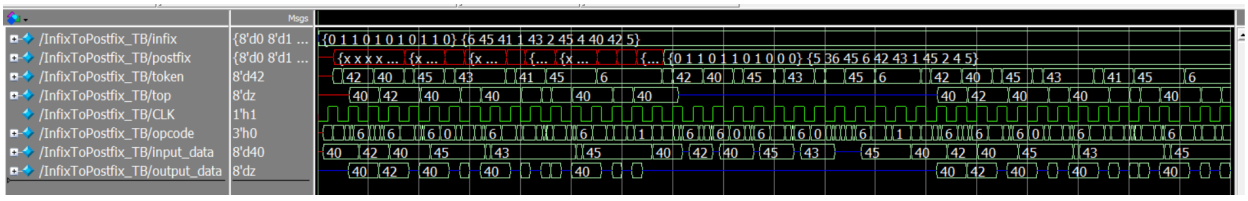
و همینطور الی آخر. اکنون بدیهی است که علت استفاده از Input[1][i] چه می‌باشد، فرض کنید که می‌خواهیم عبارت زیر را محاسبه کنیم:

40 * (42 + 41)

دقت شود که هم عدد 40 داریم و هم علامت ‘)’ که کد اسکی آن 40 است. هم عدد 42 داریم و هم علامت ‘*’ که اسکی آن 42 است. در صورت نبود چیزی مانند input[1][i] که مشخص کند بخش i ام عدد است یا کاراکتر، ما را به دردسر می‌اندازد. بنابراین از input[1][i] استفاده می‌کنیم که مشخص کند که بخش i ام رشته، عدد (operand) است یا یک (‘(‘, ‘)’, operand). که اگر input[i][1] = 0 یعنی operand در غیراینصورت یعنی operator یا پرانتز.

دقت شود که پس از تبدیل کامل infix به postfix، برای اینکه انتهای آن مشخص باشد از یک علامت ‘$’ استفاده می‌کنم.

باقی موضوعات صرفا به پیاده سازی الگوریتم (که در لینک داده شده ذکر شده) و همچنین بحث زمانبندی برمی‌گردد که با سختی و تلاش و کوشش بسیار (!) می‌توان به زمان بندی صحیح رسید. حال برای نشان دادن صحت کار این ماژول و تبدیل صحیح infix به postfix از یک ماژول در فایل InfixToPostfix_TB.v استفاده می‌کنیم. تصویر waveform آن به شکل زیر است:



حال به تحلیل آن می‌پردازیم، با توجه به اینکه ورودی به شکل زیر است:

infix/i	0	1	2	3	4	5	6	7	8	9	10
Infix[0][i]	5	42=’*’	40=’(‘	4	45=’-‘	2	43=’+’	1	41=’)’	45=’-‘	6
Infix[1][i]	0	1	1	0	1	0	1	0	1	1	0

بنابر جدول بالا، عبارت مطابق زیر است:

5 * (4 – 2 + 1) – 6

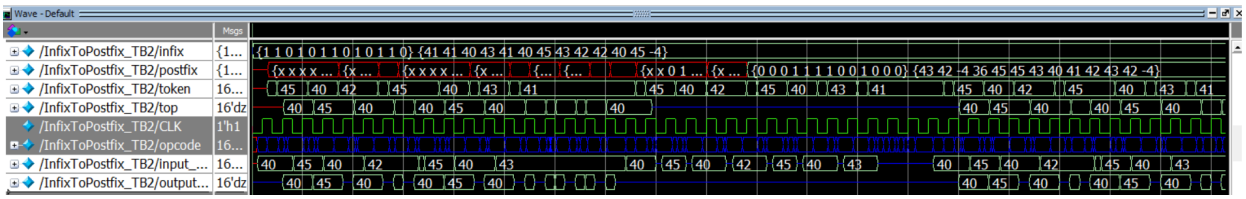
حال به خروجی postfix می‌پردازیم تا ببینیم آیا تبدیل به postfix درست صورت گرفته یا خیر:

postfix/i	0	1	2	3	4	5	6	7	8	9
postfix[0][i]	5	4	2	45 = '-'	1	43='+'	42='*'	6	45='-'	36='$'
postfix[1][i]	0	0	0	1	0	1	1	0	1	1

که عبارت حاصل به شکل زیر خواهد بود:

5 4 2 – 1 + * 6 - $

که به درستی نشان‌دهنده postfix شده‌ی عبارت infix قبل است. برای بررسی تشخیص درست کد میان علموند و عمگرها از یک تست بنچ دیگر نیز استفاده می‌کنم. فایل InfixToPostfix_TB2.v یک تست دیگر برای بررسی صحت کارکرد است. Waveform آن بدین صورت است:



حال بررسی ورودی و خروجی آن را نیز انجام می‌دهیم.

infix/i	0	1	2	3	4	5	6	7	8	9	10	11	12
Infix[0][i]	-4	45='-'	40='('	42	42='*'	43	45='-'	40='('	41	43='+'	40	41=')'	41=')'
Infix[1][i]	0	1	1	0	1	0	1	1	0	1	0	1	1

-4 – (42 * 43 – (41 + 40))

postfix/i	0	1	2	3	4	5	6	7	8	9
postfix[0][i]	-4	42	43	42 = '*'	41	40	43='+'	45='-'	45='-'	36='$'
postfix[1][i]	0	0	0	1	0	0	1	1	1	1

-4 42 43 * 41 40 + - - $

عبارت infix و postfix مربوطه مطابق بالا است.

حال که از صحت عملکرد ماژول InfixToPostfix مطمئن شدیم، وقت معرفی ماژولی است که وظیفه استفاده از این ماژول و تبدیل postfix به نتیجه نهایی عبارت ریاضی را دارا است که آن، ماژول MathSolver است. این ماژول در فایل MathSolver.v موجود است.

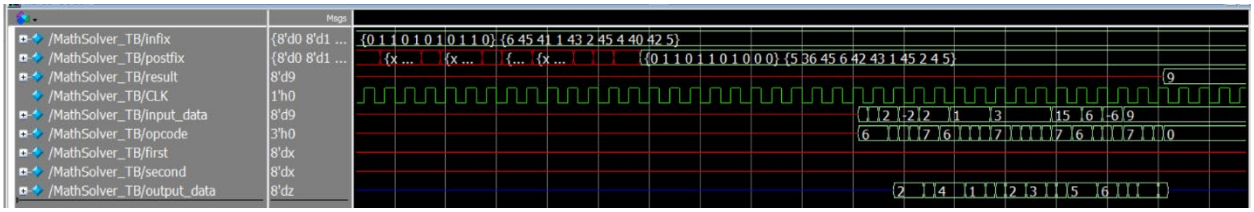
این ماژول با استفاده از ماژول InfixToPostfixModule (همان ماژول InfixToPostfix است اما در فایل InfixToPostfixModule.v قرار گرفته است و پورت‌های خروجی‌ای که برای دیباگ کردن در آن قرار داده شده بود حذف شدند تا شلوغی کار کاسته شود.) ابتدا infix را به postfix تبدیل کرده و در نهایت خروجی postfix را به یک STACK_BASED_ALU می‌دهد تا محاسبات انجام شوند. در خود MathSolver، نحوه تبدیل postfix به حاصل ریاضی پیاده سازی شده است که اینگونه است که در postfix جلو رفته و هر operand ای که می‌بینیم در stack پوش می‌کنیم تا زمانی که به یک operand برسیم و در اینصورت، دو عضو بالایی استک را پاپ کرده و این operand را روی آن دو اعمال می‌کنیم و حاصل را دوباره در استک پوش می‌کنیم. اینکار را تا زمانی انجام می‌دهیم که به $ برسیم و در نهایت، آخرین عددی که در stack باقی می‌ماند، حاصل عبارت ریاضی ماست.

این ماژول نیز زمانبندی خاص خود را دارد و از همان تکنیک 10k + 5 (زمان ریست) و 10k’ (زمان کار) تبعیت می‌کند که قبلا توضیح داده شده است. برای بررسی صحت عملکرد این ماژول، 3 عبارت ریاضی را محاسبه می‌کنیم، دو تا از آنها، همان‌هایی هستند که در بالا نوشته شده بودند و یکی هم خود عبارت صورت سوال.

اولین فایل تست بنچ، فایل MathSolver_TB.v است که رابطه ریاضی

5 * (4 – 2 + 1) – 6

را محاسبه می‌کند که قبل تبدیل شده postfix آن را محاسبه کرده بودیم:

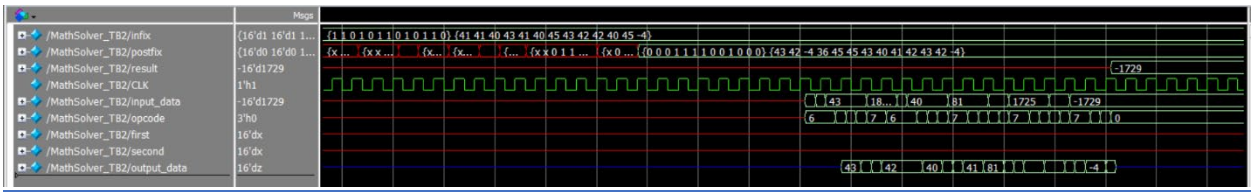


همانطور که مشاهده می‌شود، سیگنال result برابر 9 شده که همان حاصل عبارت بالا است.

حال دومین تست را انجام می‌دهیم که در فایل MathSolver_TB2.v موجود است و همان عبارت

-4 – (42 * 43 – (41 + 40))

است.



همانطور که مشاهده می‌شود، خروجی نهایی در result قابل رویت است که برابر 1729- شده است.

نهایتا آخرین تست این سوال را که همان تست صورت سوال است انجام می‌دهم که در فایل MathSolver_TB3.v موجود است و عبارت:

2 * 3 + (10 + 4 + 3) * -20 + (6 + 5)

است.

دقت شود که 323- همان حاصل عبارت است که در result دیده می‌شود.

سوال 6:

سؤال ۶:

یک ماژول وریلاگ برای فلیپ‌فلاپ نوع D طراحی کنید و با استفاده از آن یک شمارنده حلقوی جانسون N بیتی طراحی بسازید. مدار خود را برای Nهای ۴، ۸، ۱۶ و ۳۲ مورد آزمون قرار دهید. دقت کنید در این سؤال از ویژگی پارامتر برای تعیین N و دستور generate برای ساخت شمارنده با N فلیپ‌فلاپ نوع D استفاده کنید (۵۰ نمره).

در ابتدا برای ساخت DFF خود اقدام می‌کنیم. من یک DFF با سیگنال‌های SET, RESET **سنکرون** ساختم. اسم فایل کد این ماژول، DFF.v می‌باشد.

جدول زیر نشان‌دهنده انتظار من از این DFF در حالات ورودی مختلف است.

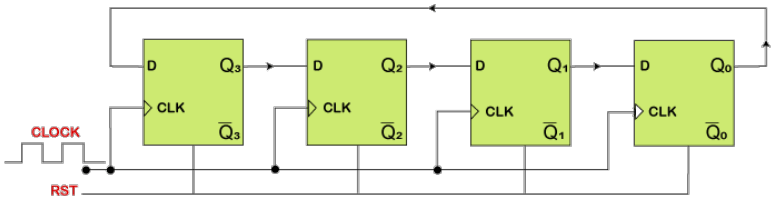
CLK	SET	RESET	D	Q
↑	1	x	x	1
↑	0	1	x	0
↑	0	0	0	0
↑	0	0	1	1

برای این ماژول یک تست بنچ نیز طراحی کردم تا از صحت عملکرد DFF خود مطمئن شوم. این تست بنچ در فایل DFF_TB.v موجود است.

تصویر waveform بالا با جدول داده شده تطابق دارد بنابراین، DFF ما عملکردی صحیح دارد.

حال به ماژول اصلی خود، یعنی JohnsonRingCounter.v می‌رسیم که در پیاده سازی آن، از پارامتر n برای مشخص کردن تعداد بیت‌های شمارنده و از generate برای ساختن n تا DFF و دادن ورودی‌ها و خروجی‌های آنها استفاده شده است.

تنها نکته حائز اهمیت این است که طبق شکل زیر، در شمارنده حلقوی جانسون، تنها فلیپ فلاپ اول استثنا است و ورودی D آن از Q’ فلیپ فلاپ آخر بدست می‌آید و جز آن، هر فلیپ فلاپ دیگر، ورودی D آن از Q فلیپ فلاپ قبلی‌اش بدست می‌آید.

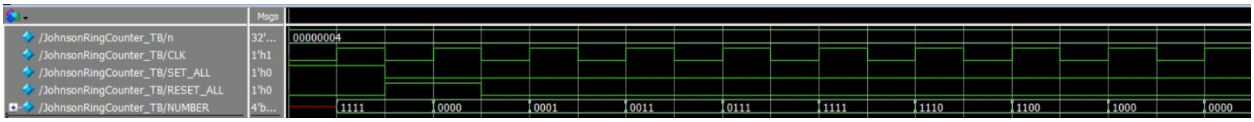


ماژول JohnsonRingCounter من، دارای ورودی‌های CLK, SET_ALL, RESET_ALL بوده و خروجی NUMBER را دارد. ورودی SET_ALL اگر 1 شود، تمام بیت‌های NUMBER، یک می‌شوند و اگر RESET_ALL یک شود و SET_ALL، صفر باشد، تمام بیت‌های NUMBER برابر 0 می‌شود. در حالاتی که SET_ALL و RESER_ALL هر دو صفر باشند، شمارش شمارنده جانسون شروع می‌شود.

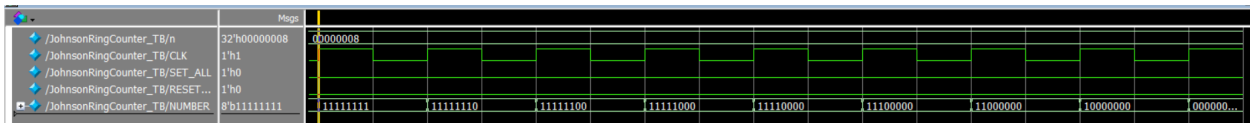
به منظور نشان‌دادن صحت عملکرد این شمارنده، یک تست بنچ در فایل JohnsonRingCounter_TB.v موجود است که با پارامتر n کار می‌کند و برای تست کردن مقادیر مختلف n، کافی است عدد آن را عوض کنیم. من این اعداد را برابر 4، 8، 16 و 32 قرار می‌دهم.

در این تست بنچ، در ابتدا، SET_ALL = 1 قرار دادم و طبعا تمام بیت‌های شمارنده باید 1 شوند. پس از آن RESET_ALL = 1 و SET_ALL = 0 کردم تا دوباره تمام بیت‌ها 0 شوند. پس از آن RESET_ALL = 0 و SET_ALL = 0 کردم تا شمارش به طور عادی شروع شود.

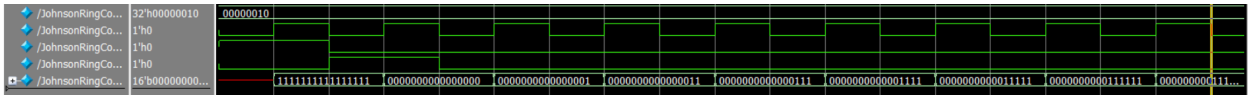
تصاویر شمارش 4 بیتی:

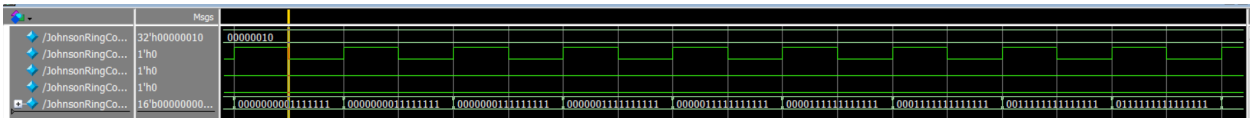


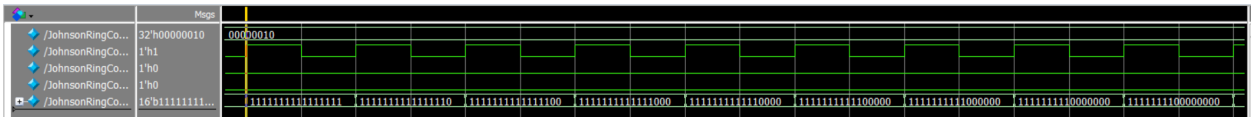
تصاویر شمارش 8 بیتی:

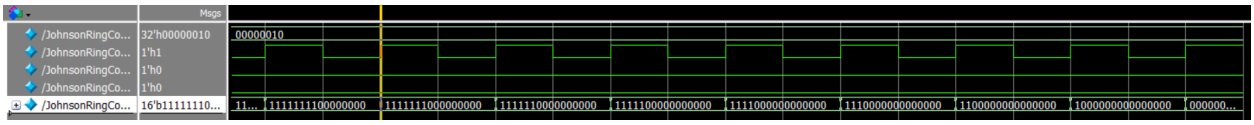


تصاویر شمارش 16 بیتی:

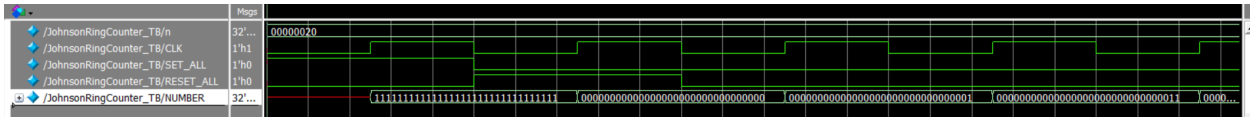


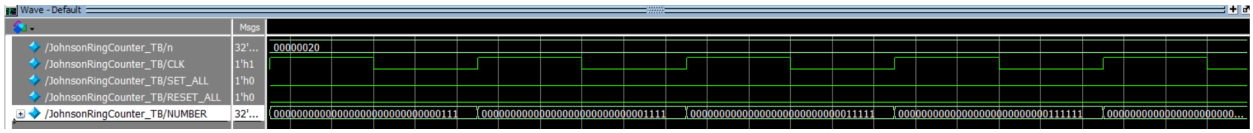






تصاویر شمارش 32 بیتی:





تصاویر با همین روند ادامه دارند، از گذاشتن عکس‌های بیشتر خودداری کرده و صرفا عکس نهایی را می‌گذارم: